APS105
Winter 2012

Jonathan Deber
jdeber -at- cs -dot- toronto -dot- edu

Lecture 27
March 23, 2012

Today

- Recursion
- `struct`

Recursion

In order to understand recursion,
you must first understand recursion.

Anonymous

Recursion

- A problem defined in terms of itself
- No more (or less) powerful than loops (iteration)

If I need to put away a pile of books:

Put away the first book

Put away the remaining pile

Put away the books one at a time

If I need to climb a flight of stairs:

Climb one stair

Climb the remaining flight of stairs

Climb the stairs one at a time

If I need to calculate n!

Multiply `n * (n - 1)!`

Start at 1 and multiply it by each number up to n

Recursion

- A problem defined in terms of itself
- No more (or less) powerful than loops (iteration)
- All recursive problems must have a “base case”, which is a trivial version of the problem

If I need to put away a pile of books:

Put away the first book

Put away the remaining pile

If I need to climb a flight of stairs:

Climb one stair

Climb the remaining flight of stairs

If I need to calculate n!

Multiply `n * (n - 1)!`

If I need to put away a pile of books:

If there are no more books, we're done

Otherwise

Put away the first book

Put away the remaining pile

If I need to climb a flight of stairs:

If we're at the top, we're done

Otherwise

Climb one stair

Climb the remaining flight of stairs

If I need to calculate n!

If n is 1,

n! is 1

Otherwise

Multiply `n * (n - 1)!`

Recursion and Programming

- A recursive function is a function that calls itself
- That's it!
- Provides another way to iterate
- No more (or less) powerful than loops
  - Similar to `for` loops vs. `while` loops
- Why bother? Makes some algorithms simpler

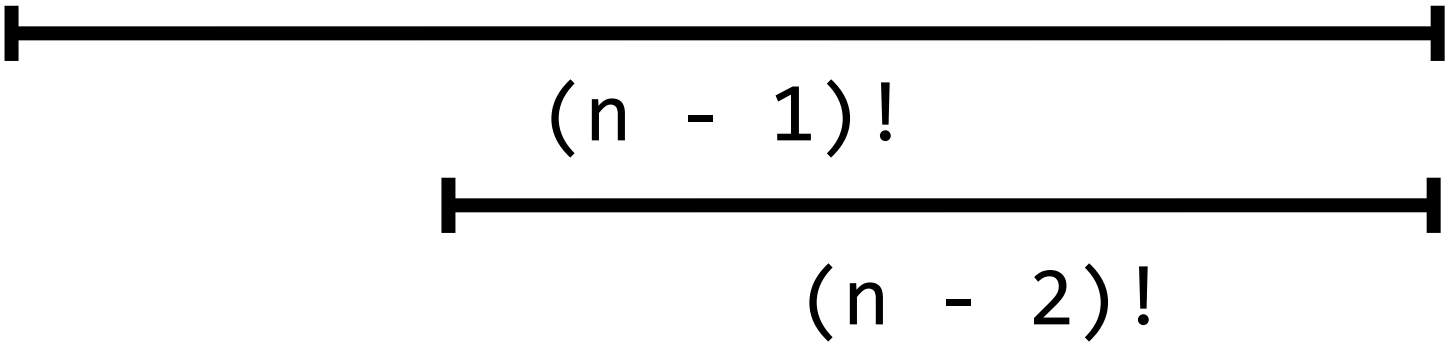
Factorial

n! is n * (n - 1) * (n - 2) * ... * 2 * 1

```
int factorialIterative(int n)
{
    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }

    return result;
}
```

Factorial

n! is n * (n - 1) * (n - 2) * ... * 2 * 1



n! is n * (n - 1)!

```
int factorialRecursive(int n)
{
  int result;

  if (n == 1)
  {
    result = 1;
  }
  else
  {
    result =  n * factorialRecursive(n - 1);
  }
  return result;
}
```

Fibonacci

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0,\ F(1) = 1$$

0, 1, 1, 2, 3, 5, 8, 13, 21,
34, 55, 89, 144, 233, 377, 610,
987, 1597, 2584, 4181, ...

```
int fibonacciIterative(int n)
{
    int fib_n = 1;         // fib(1)
    int fib_nMinus1 = 0; // fib(0)

    for (int i = 2; i <= n; i++)
    {
        int nextTerm = fib_n + fib_nMinus1;

        fib_nMinus1 = fib_n;
        fib_n = nextTerm;
    }

    int result;

    if (n == 0)
    {
        result = fib_nMinus1;
    }
    else
    {
        result = fib_n;
    }

    return result;
}
```

```
int fibonacciRecursive(int n)
{
    int result;

    if (n == 0)
    {
        result = 0;
    }
    else if (n == 1)
    {
        result = 1;
    }
    else
    {
        result = fibonacciRecursive(n - 1)
                        + fibonacciRecursive(n - 2);
    }

    return result;
}
```

```
int fibonacciIterative(int n)
{
  int fib_n = 1;        // fib(1)
  int fib_nMinus1 = 0; // fib(0)

  for (int i = 2; i <= n; i++)
  {
    int nextTerm = fib_n + fib_nMinus1;

    fib_nMinus1 = fib_n;
    fib_n = nextTerm;
  }

  int result;

  if (n == 0)
  {
    result = fib_nMinus1;
  }
  else
  {
    result = fib_n;
  }

  return result;
}
```

```
int fibRecursive(int n)
{
  int result;

  if (n == 0)
  {
    result = 0;
  }
  else if (n == 1)
  {
    result = 1;
  }
  else
  {
    result = fibRecursive(n - 1)
             + fibRecursive(n - 2);
  }

  return result;
}
```

Recursive Structure

- Steps for writing a recursive algorithm:
  - Figure out the recursive structure of the problem
    - Must be able to break the problem into two parts
    - Each must be either trivial or a smaller version of the same problem
  - Figure out the trivial base case(s)
  - Convert pseudocode to C

Print out a line of n stars

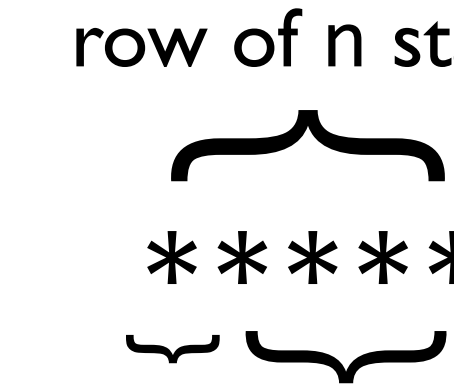
```
n = 1      *
n = 5      *****
```

```
void printRow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("*");
    }
}
```

Print out a line of n stars

n = 1 *

n = 5 *****



If n is 0

Do nothing

Otherwise

Print out 1 star

Print out a row of n - 1 stars

```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
  else
  {

  }
}
```

If n is 0

Do nothing

Otherwise

Print out 1 star

Print out a row of n - 1 stars

```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
}
```

If n is 0

Do nothing

Otherwise

Print out 1 star

Print out a row of n - 1 stars

```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
}
```

```
if (n > 1)
{
  printf("*");
  printRow(n - 1);
}
else
{
  printf("*");
}
```

If n is 0

Do nothing

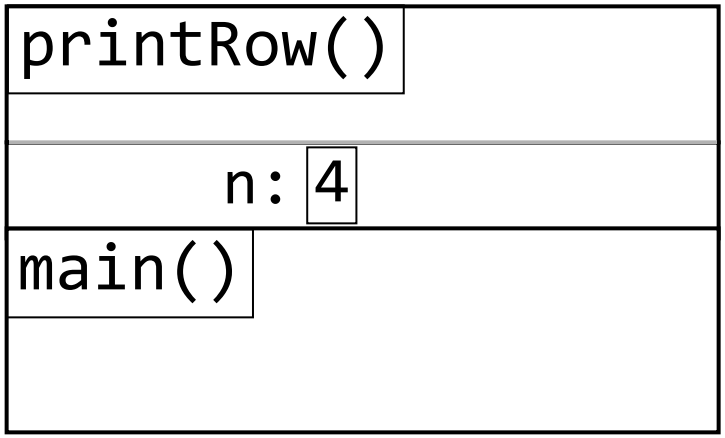
Otherwise

Print out 1 star

Print out a row of n - 1 stars

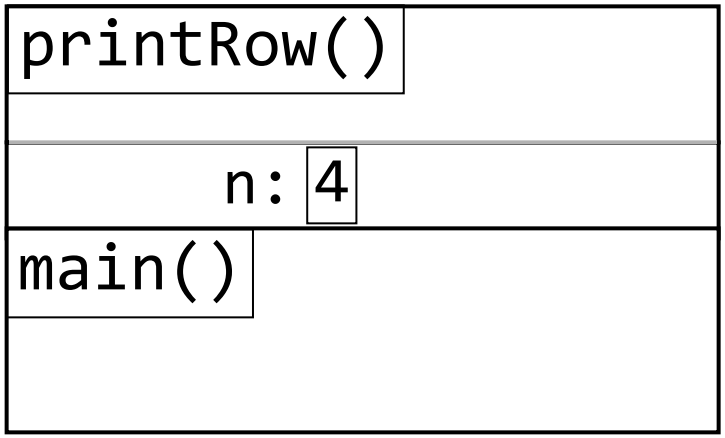
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```



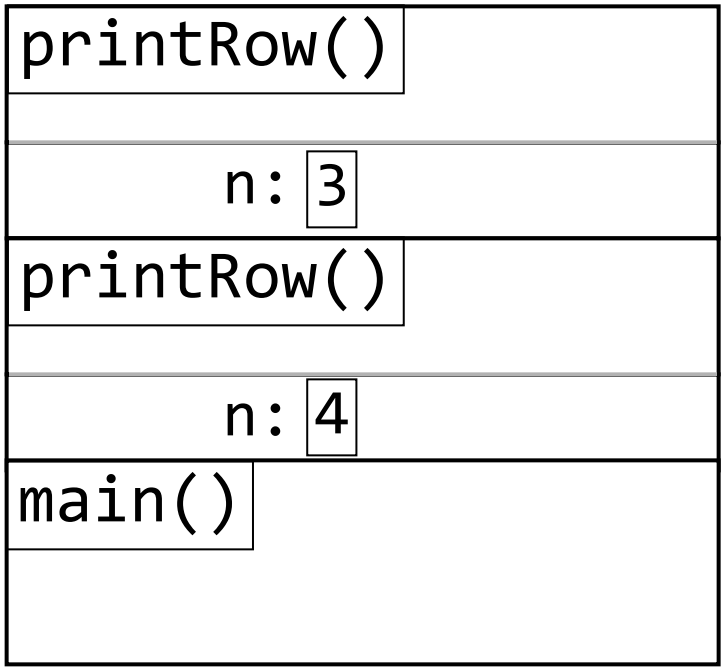
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
*
```



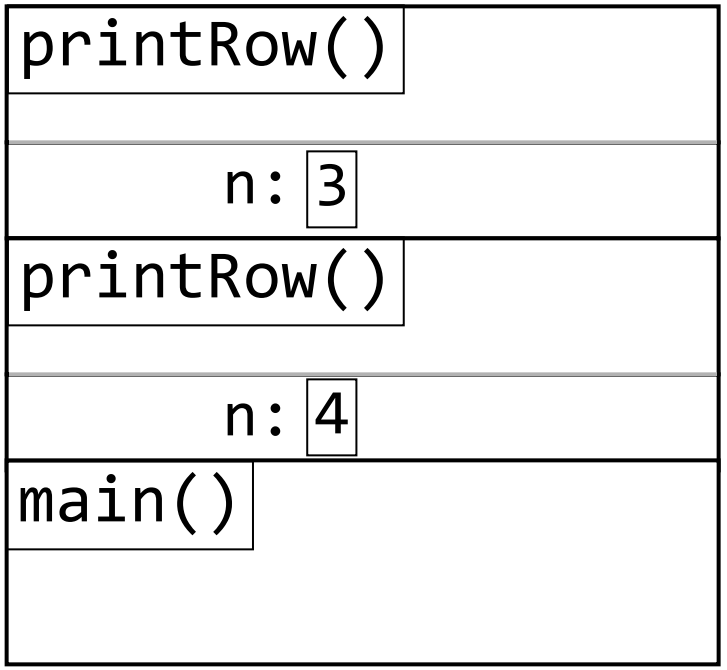
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
*
```



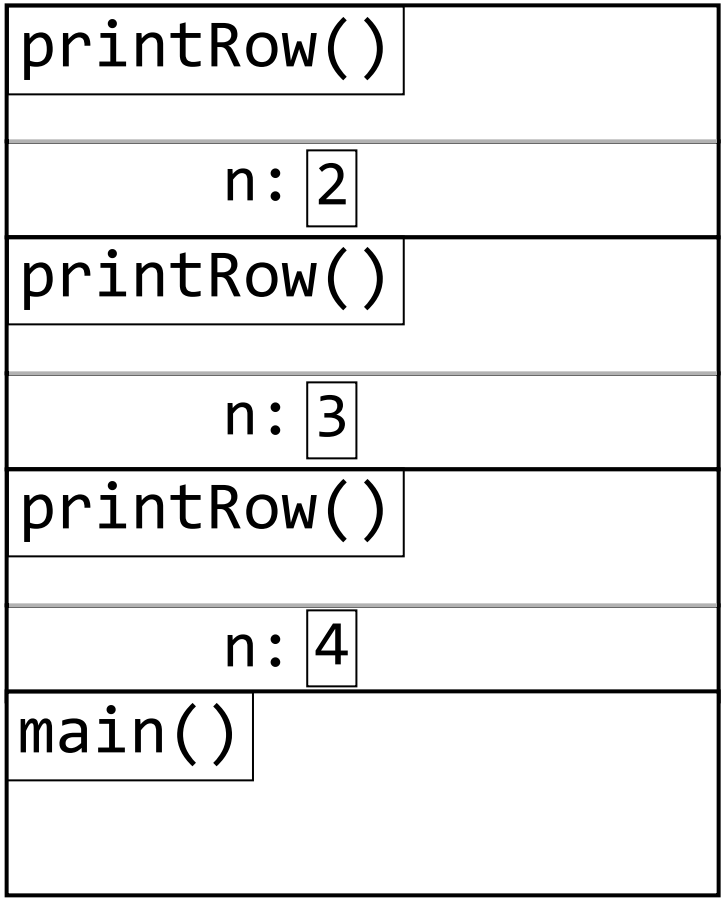
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* *
```



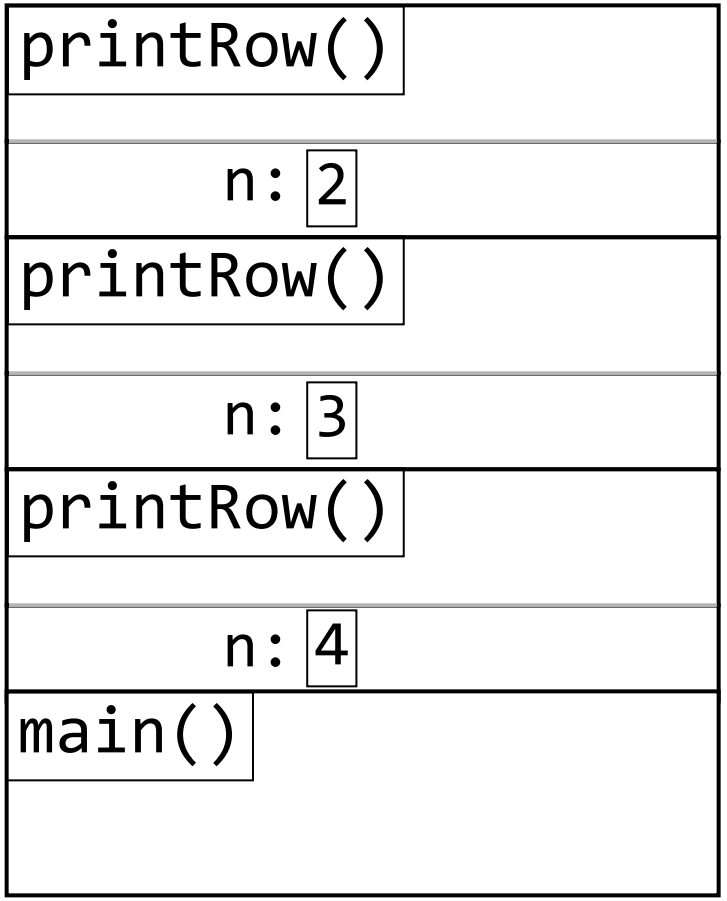
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* *
```



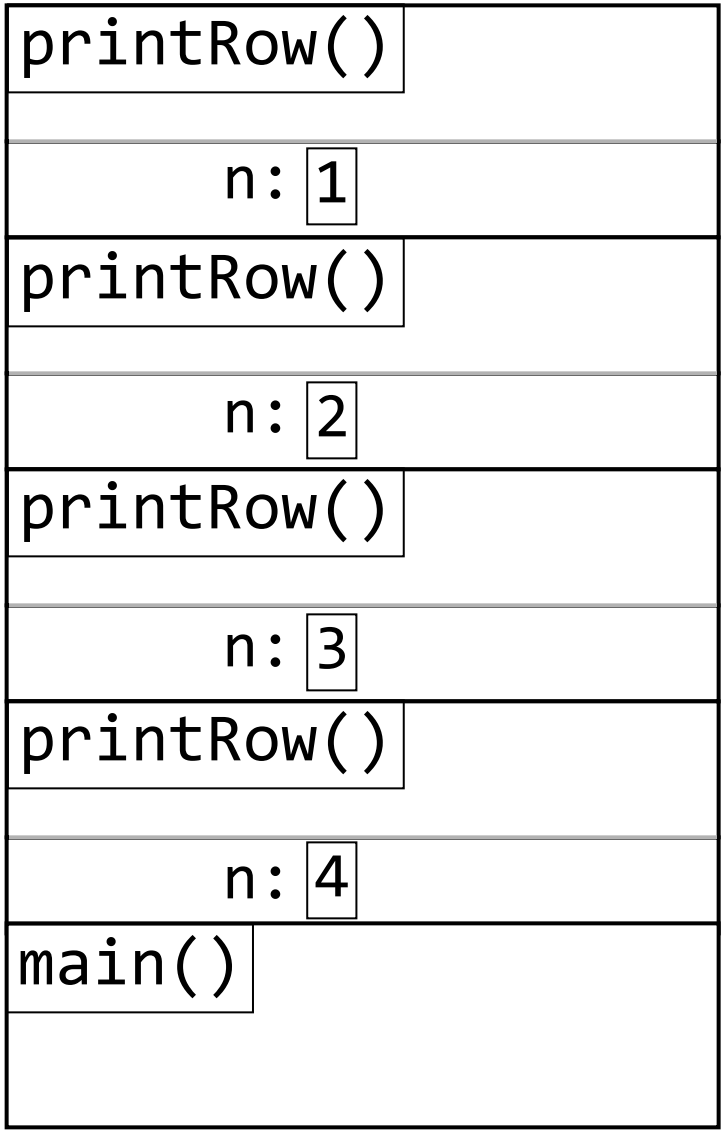
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
***
```



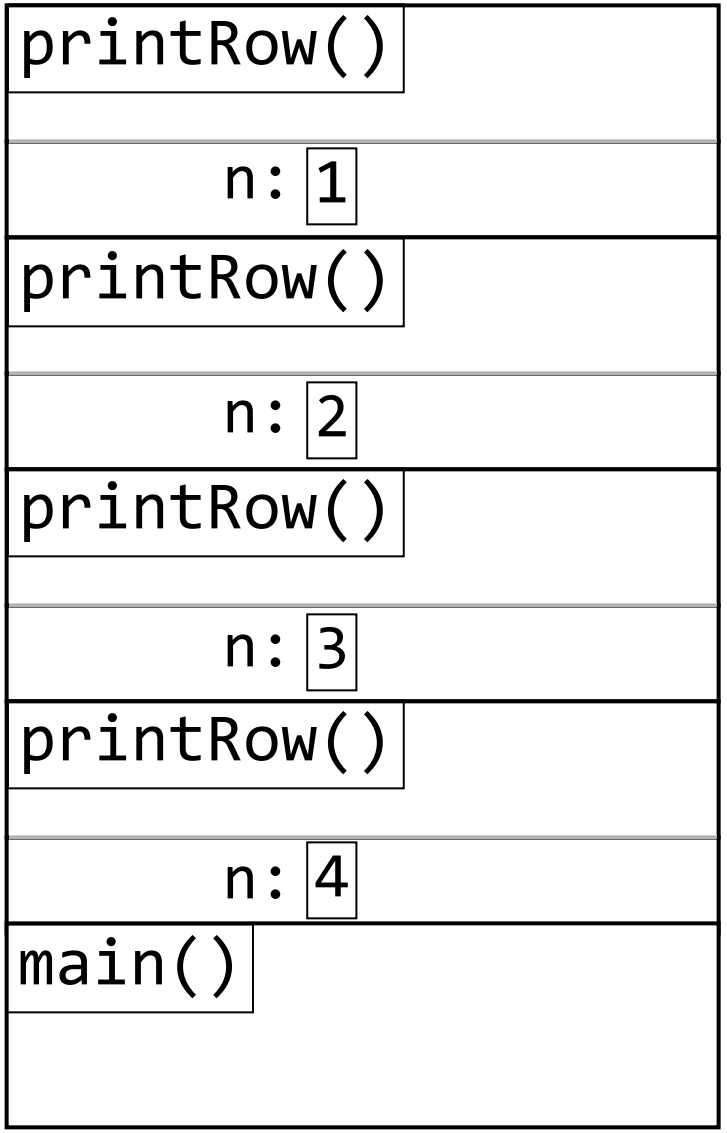
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

* * *



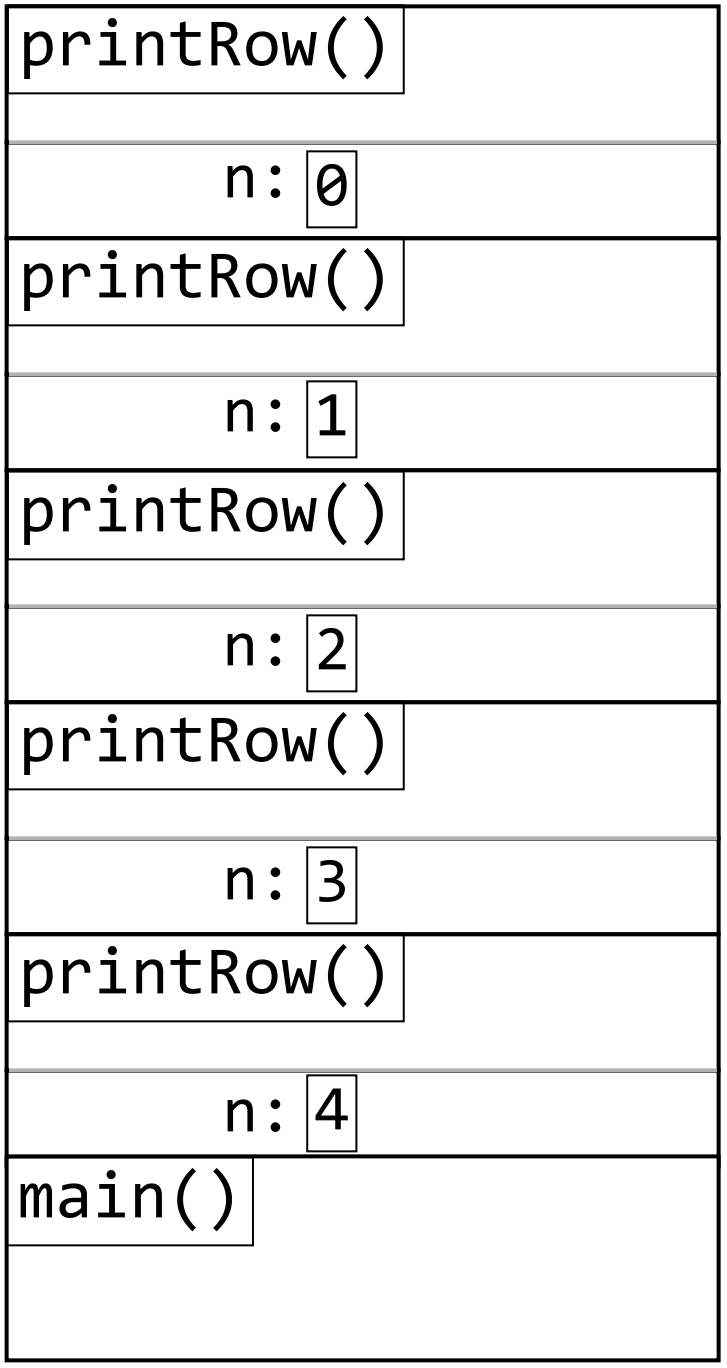
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```



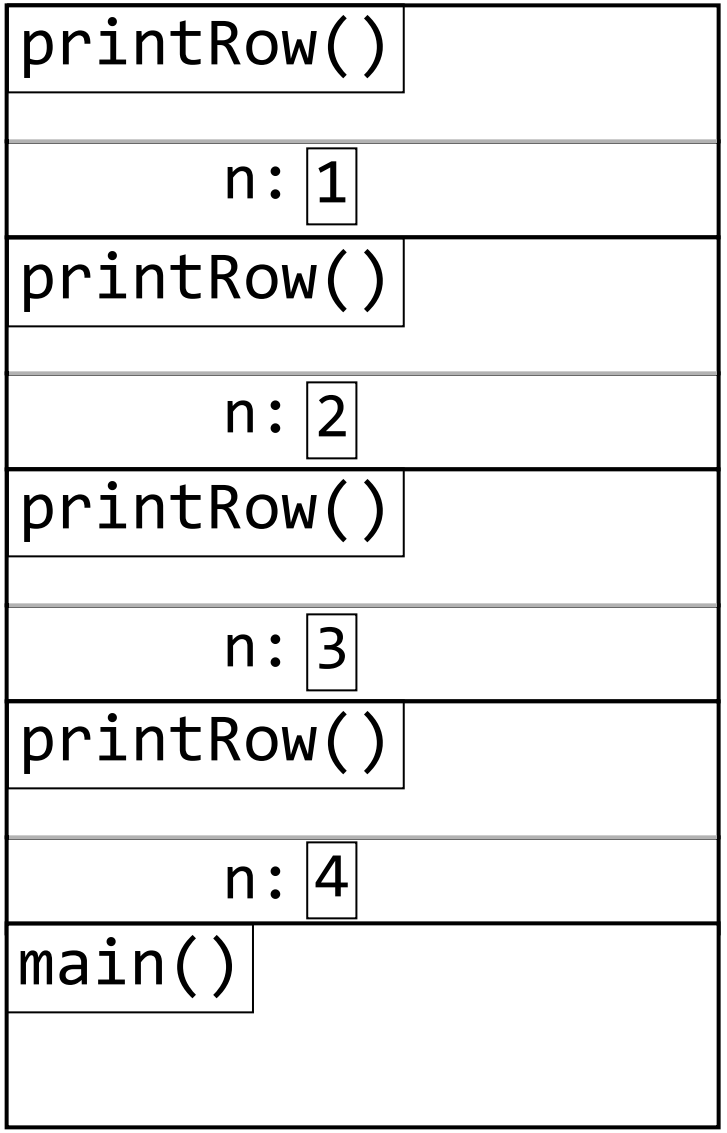
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```



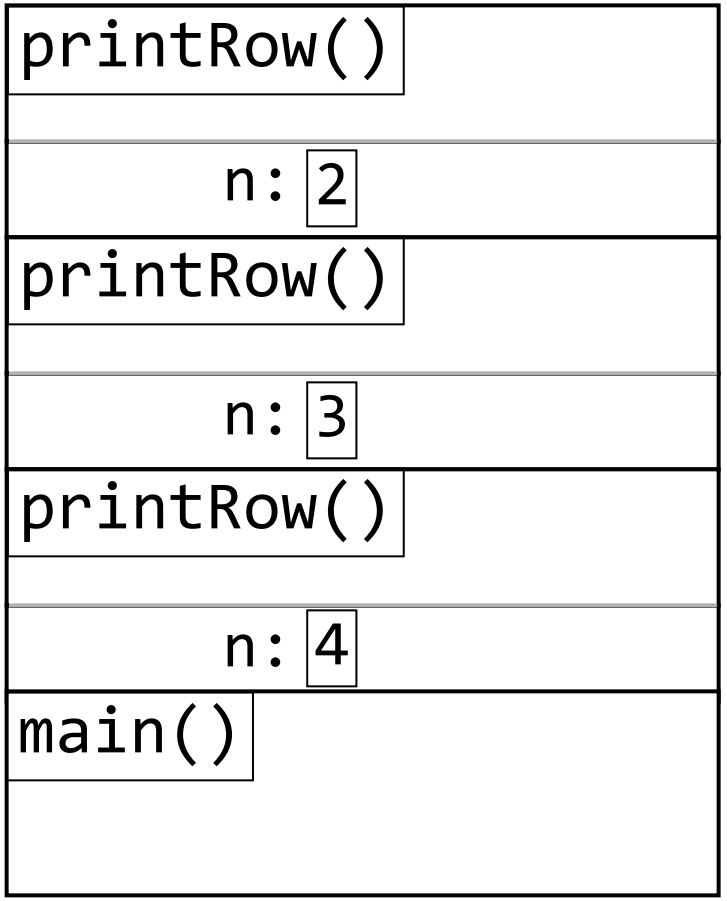
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```



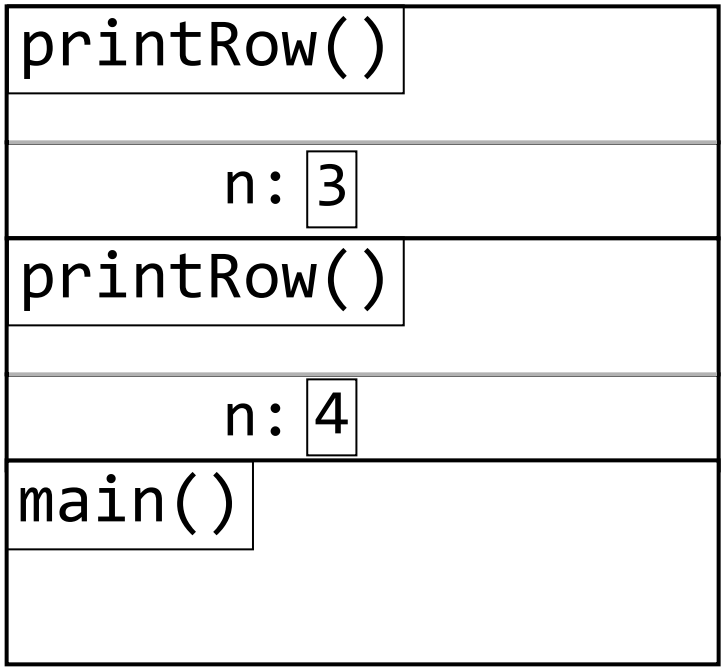
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```



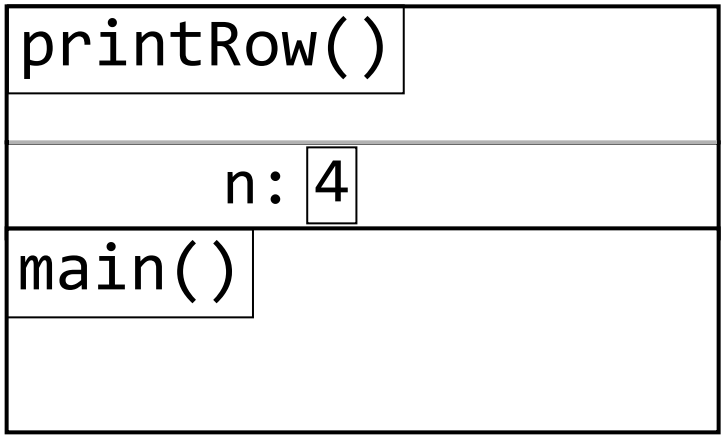
```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```



```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```



```
void printRow(int n)
{
  if (n > 0)
  {
    printf("*");
    printRow(n - 1);
  }
 }

int main(void)
{
  printRow(4);

  return 0;
}
```

```
* * * *
```

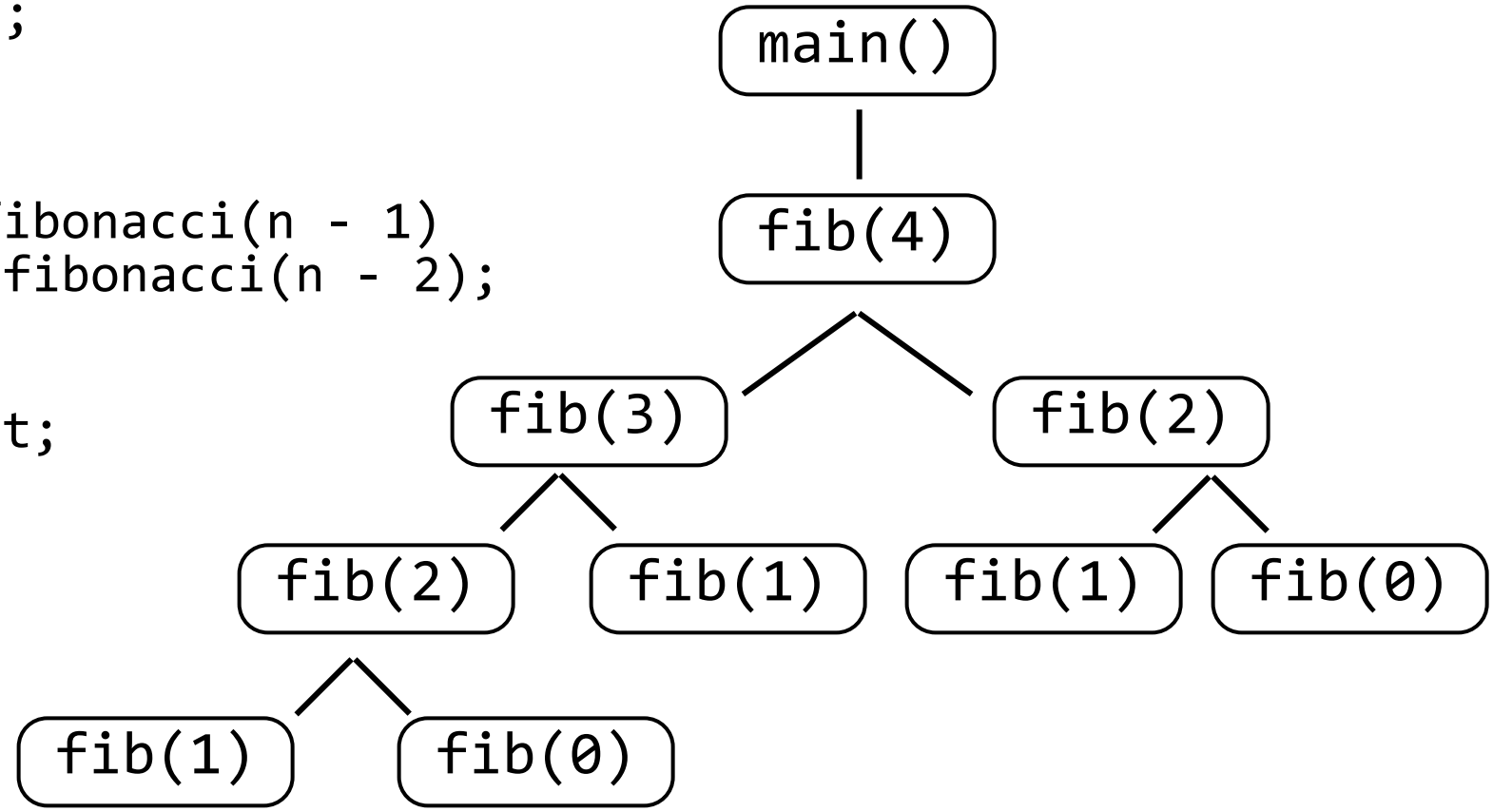
```
int fibonacci(int n)
{
  int result;

  if (n == 0)
  {
    result = 0;
  }
  else if (n == 1)
  {
    result = 1;
  }
  else
  {
    result = fibonacci(n - 1) + fibonacci(n - 2);
  }

  return result;
}
```

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0, F(1) = 1$$

```
int fibonacci(int n)
{
  int result;

  if (n == 0)
  {
    result = 0;
  }
  else if (n == 1)
  {
    result = 1;
  }
  else
  {
    result = fibonacci(n - 1)
           + fibonacci(n - 2);
  }

  return result;
}
```

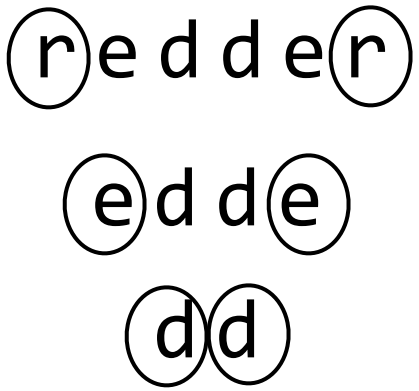
```
int main(void)
{
  int f = fibonacci(4);

  return 0;
}
```



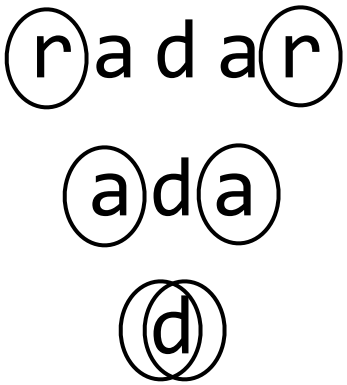
Function Call	*Number of Calls*
`fibonacci(15)`	1
`fibonacci(14)`	1
`fibonacci(13)`	2
`fibonacci(12)`	3
`fibonacci(11)`	5
`fibonacci(10)`	8
`fibonacci(9)`	13
`fibonacci(8)`	21
`fibonacci(7)`	34
`fibonacci(6)`	55
`fibonacci(5)`	89
`fibonacci(4)`	144
`fibonacci(3)`	233
`fibonacci(2)`	377
`fibonacci(1)`	610
`fibonacci(0)`	377

Efficiency

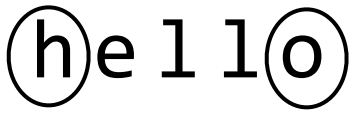
- With recursion, it's possible to end up doing the same work over and over again
- This can also happen with iteration, but it's usually more obvious
- There are ways to avoid it, but things get slightly more complicated



yes



yes



no

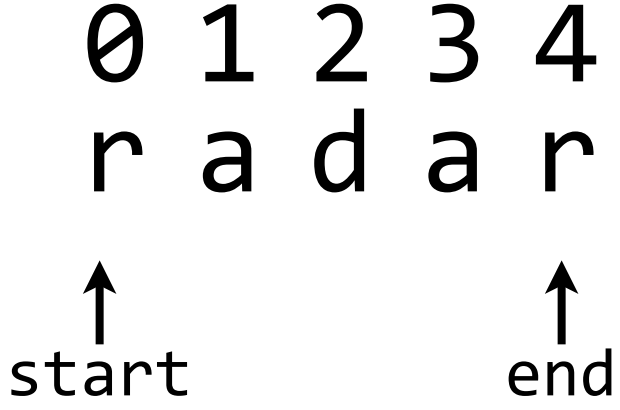
```
if (length is 0 or 1)
    It's a palindrome
Otherwise
    if (the first and last characters match)
        See if remaining string is a palindrome
    else
        It's not a palindrome
```

```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```

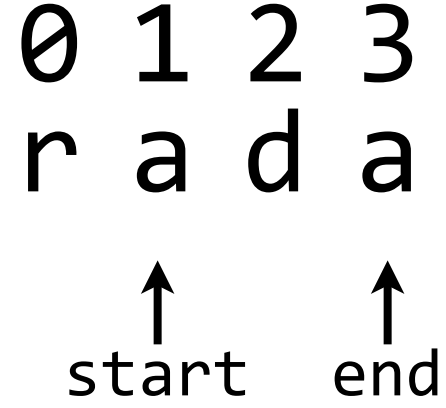
if (length is 0 or 1)
  It's a palindrome

Otherwise
  if (the first and last characters match)
    See if remaining string is a palindrome
  else
    It's not a palindrome

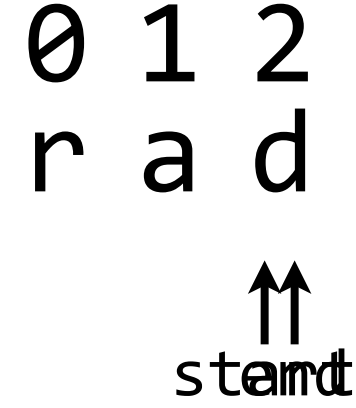
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



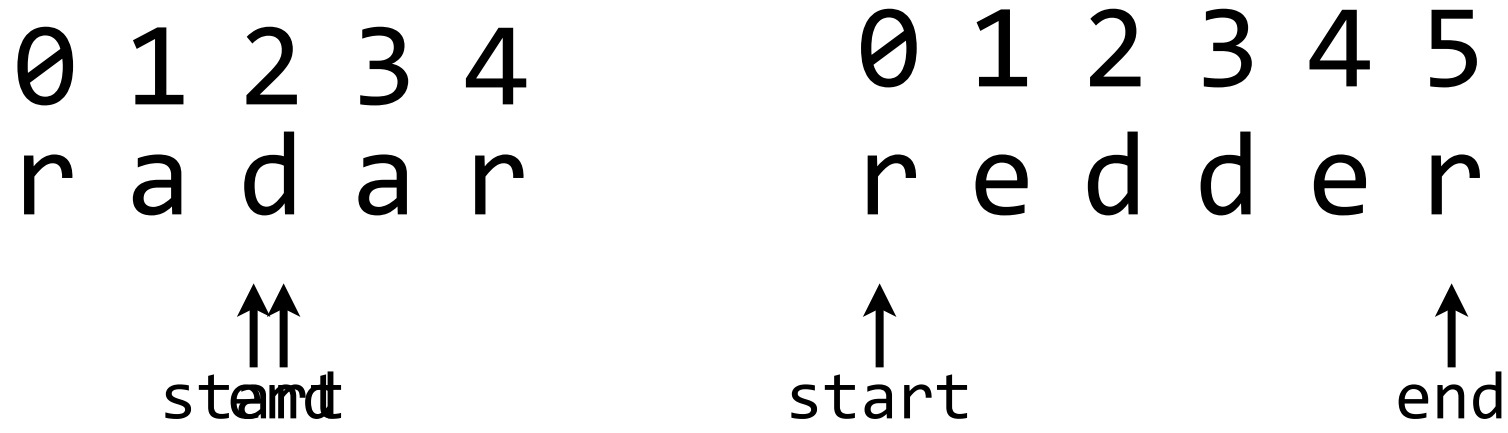
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



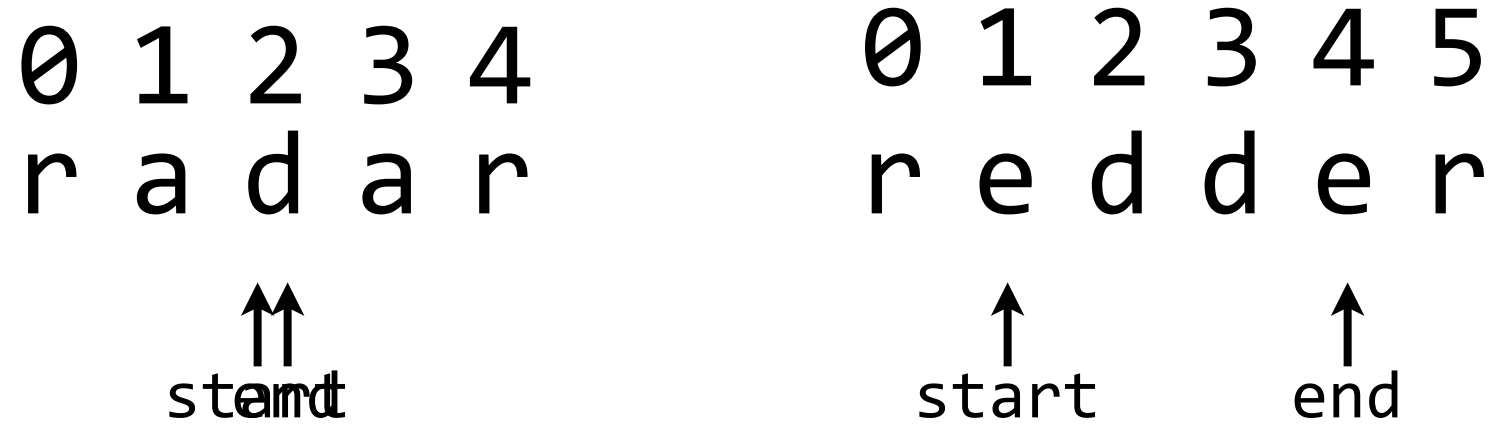
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



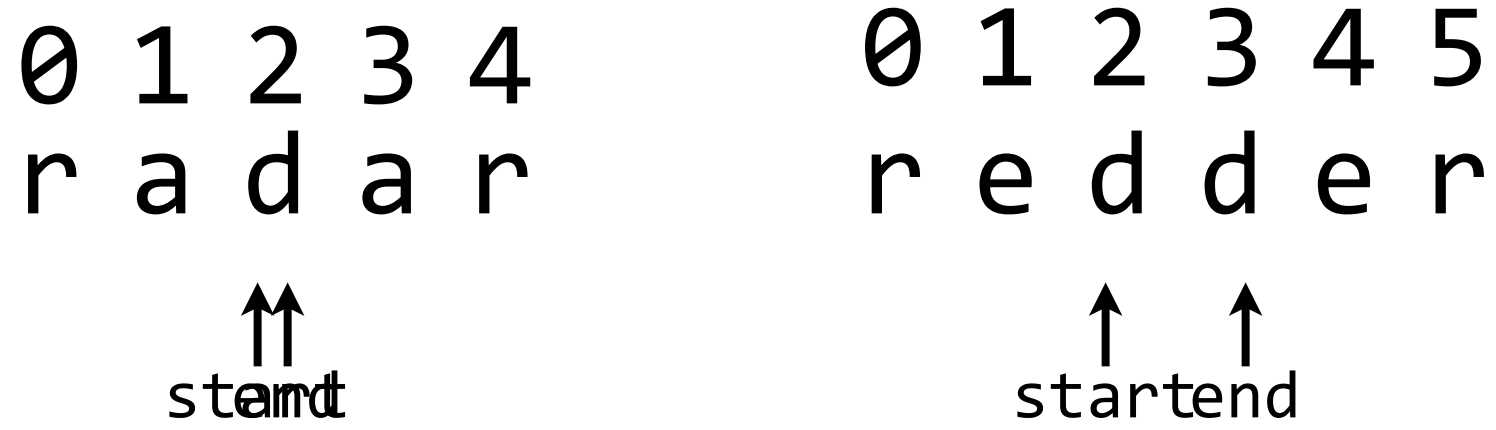
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



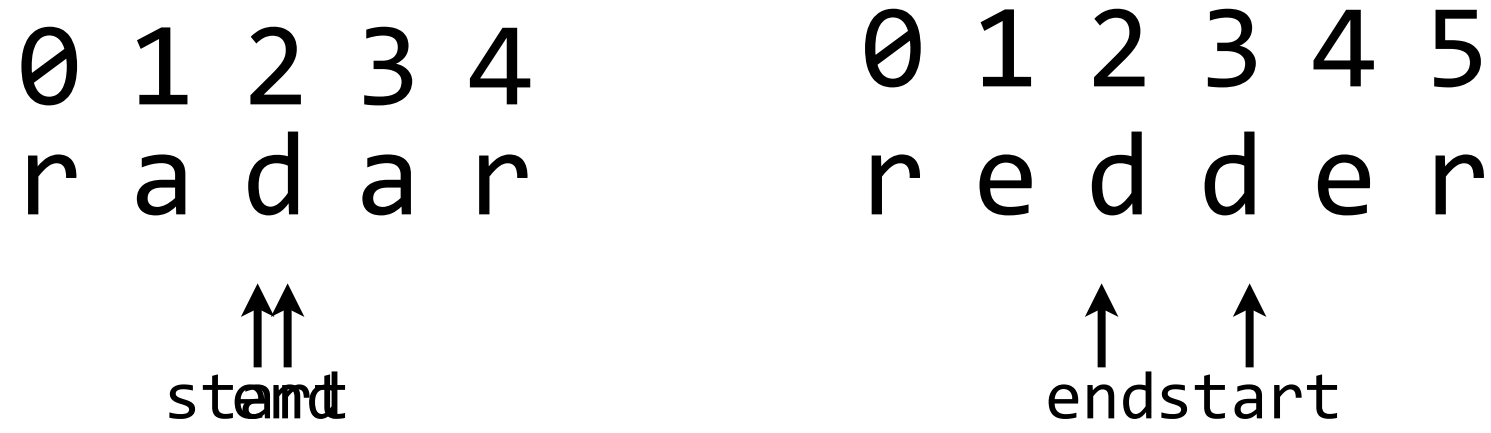
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



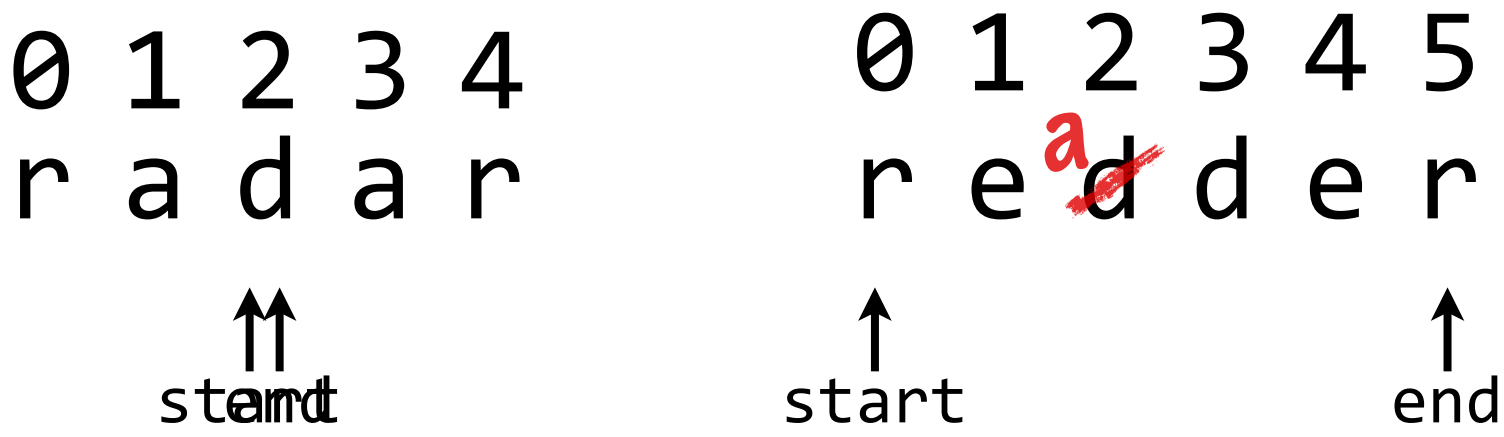
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



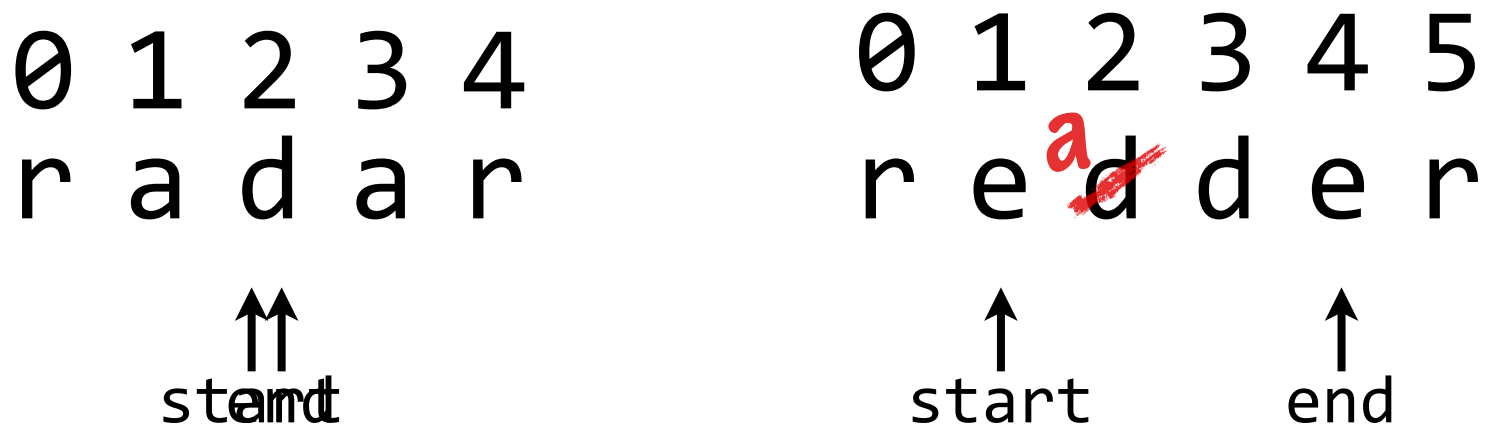
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



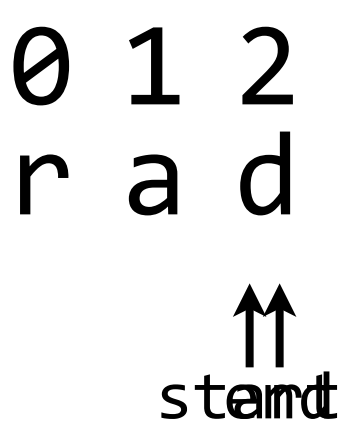
```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)                    if (end - start <= 1)?
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```

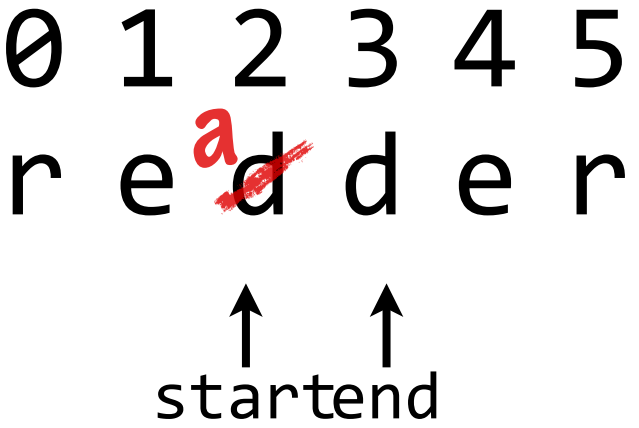


```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)                    if (end - start <= 1)?
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```



```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)                    if (end - start <= 1)?
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}
```





```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}


char *s = "radar";
bool palindrome = isPalindrome(s, 0, strlen(s) - 1);
```

```
bool isPalindrome(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindrome(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}

bool isPalindrome(char *s)
{
  return isPalindrome(s, 0, strlen(s) - 1);
}

char *s = "radar";
bool palindrome = isPalindrome(s);
```

Error

```
bool isPalindromeHelper(char *s, int start, int end)
{
  // If there are 0 or 1 characters left
  if (end - start < 1)
  {
    return true;
  }
  // If the first and last character match
  else if (s[start] == s[end])
  {
    return isPalindromeHelper(s, start + 1, end - 1);
  }
  else
  {
    return false;
  }
}

bool isPalindrome(char *s)
{
  return isPalindromeHelper(s, 0, strlen(s) - 1);
}

char *s = "radar";
bool palindrome = isPalindrome(s);
```

Structures

Data

```
int i = 9;

int i_0 = 0;
int i_1 = 1;
int i_2 = 2;
int i_3 = 3;

int a[N];

for (int i = 0; i < N; i++)
{
   a[i] = i;
}
```

Arrays

- Group data together
- Allow easy access to each element

- Can only hold one type of data

Name of city (char *)	Population in millions (double)
"Halifax"	0.283
"Montreal"	3.764
"Toronto"	6.324
"Vancouver"	2.254

Name of city (char *)	Population in millions (double)
"Halifax"	0.283
"Montreal"	3.764
"Toronto"	6.324
"Vancouver"	2.254

(char * and double) cities[4];

Error

Name of city (char *)	Population in millions (double)	
"Halifax"	0.283	"0.283"
"Montreal"	3.764	"3.764"
"Toronto"	6.324	"6.324"
"Vancouver"	2.254	"2.254"

(char * and double) cities[4]; Error

```
char *names[N] = {"Halifax", "Montreal",
                  "Toronto", "Vancouver"};

double metroPopulation[N] = {0.283, 3.764,
                             6.324, 2.254};
```

```
for (int i = 0; i < N; i++)
{
  printf("The pop of %s is %g million\n",
         names[i], metroPopulation[i]);
}
```

Parallel Arrays

- This technique is called a "parallel array"
- Can be useful for some things

- Can be easy to get mixed up
- More work to rearrange items

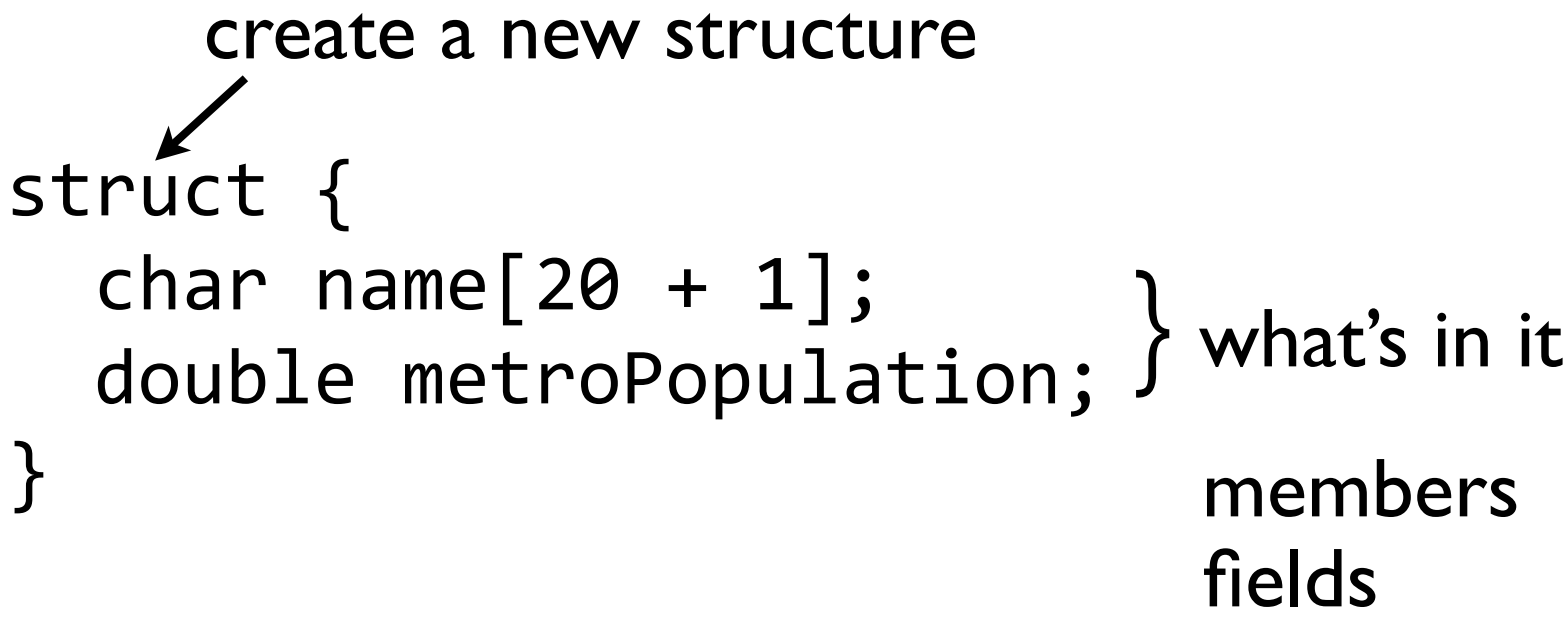
```
char *names[N] = {"Halifax", "Montreal",
                  "Toronto", "Vancouver"};

double metroPopulation[N] = {0.283, 3.764,
                             6.324, 2.254};
```

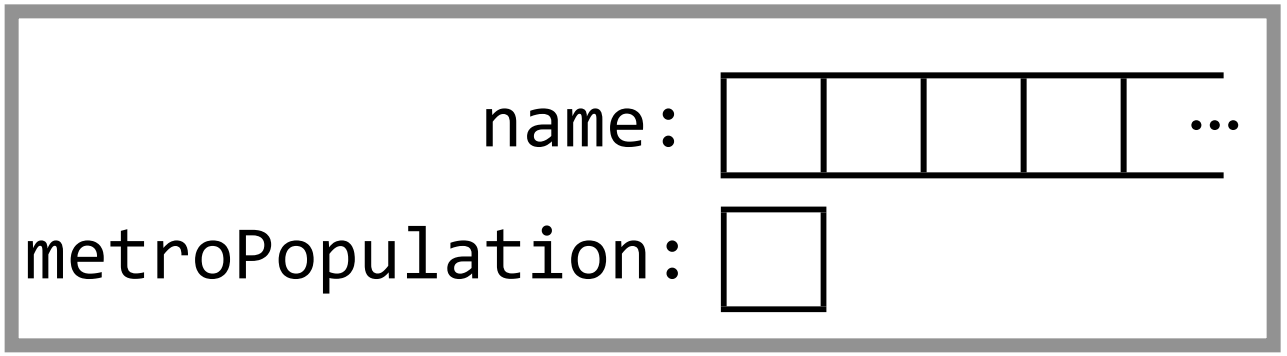
```
"Edmonton"    1.032
```

I need to move everything in both arrays over.
Twice as much to do, twice as much to get wrong.

Structures

- Way of grouping variables together into a single unit
- Acts as a template to create instances of that grouping





This code is useless on its own

```
struct {
  char name[20 + 1];                int
  double metroPopulation;
}

  ??? toronto;                  int i;
```

```
struct city {
  char name[20 + 1];
  double metroPopulation;
};
```
⟵ Watch this ;

```
struct city toronto = {"Toronto", 6.324};
```

```
int i;                      Type of i is int
int *p;                     Type of p is int *
struct city toronto;        Type of toronto is struct city
city vancouver;             Type of vancouver is city
```

Error

typedef

- A way of defining a new type
- More accurately, a new name for an existing type

```
typedef existing-type new-type;

typedef double Dollar;

Dollar price = 1.99;

Dollar calculateTax(Dollar d)
{
  ...
}
```

typedef

- We've seen several so far
  - `bool` is a `typedef` for `int`
  - `size_t` is a `typedef` for `unsigned int` (on ECF)
  - Other `_t`, e.g., `time_t`, `ptrdiff_t`, etc.
- Built-in `typedefs` usually start with lowercase letters
- By convention, our `typedefs` start with a capital

```
typedef struct city {
   char name[20 + 1];
   double metroPopulation;
} City;
```

create a `struct` named `city`

```
typedef struct city { ...  } City;
```

create a `typedef` for `struct city` called `City`

```
City toronto = {"Toronto", 6.324};
```

Type of `toronto` is `City`